Tutorials for queueing simulations

N.D. Van Foreest and E.R. Van Beesten

January 13, 2020

CONTENTS

INTRODUCTION

Typically a queueing system is subject to rules about when to allow jobs to enter the system or to adapt the service capacity. Such a decision rule is called a *policy*. The theoretical analysis of the efficacy of policies is often very hard, while with simulation it becomes doable. In this document we present a number of cases to see how simulation can be used to analyze and improve queueing systems. Besides the fact that these cases will improve your understanding of queueing systems, probability theory (such as how to compute the empirical cumulative distribution function) and (big) data analysis, they will also make clear that simulation is a really creative activity and involves solving many interesting and challenging algorithmic problems.

Each case is organized in a number of exercises. For each exercise,

1. Make a design of how you want to solve the problem. For instance, make a model of a queueing system, or a control policy structure, or compute relevant KPIs (key performance indicators, such as cost, or utilization of the server, and so on). In other words, think before you type.
2. Try to translate your ideas into pseudo code or, better yet, python[1]
3. If you don't succeed in getting your program to work, look up the code written by us and type it into your python environment.[2].
4. Simulate a number of scenarios by varying parameter settings and see what happens.

We expect you to work in a groups of 2 to 3 students and bring a laptop with an *installed and working* python environment, preferably the anaconda package available at `https://www.anaconda.com/`, as this contains all functionality we will need[3].

Note that the code is part of the course, hence of the midterms and the exams. Unless indicated as not obligatory, you have to be able to read the code and understand it. Our code is not the fastest or most efficient, rather, we focus on clarity of code so that the underlying reasoning is as clear as possible. Once our ideas and code are correct, we can start optimizing, if this is necessary.

The subsections below provide some extra information regarding the use of Python in this course. Please read it carefully.

PYTHON BACKGROUND

For the computer simulation assignments in this course it is important that you have a basic understanding of the programming language *Python*. Python is arguably easier to learn than R, which you already know, but you have to get used to the syntax. Therefore, in order to get started with Python, we strongly advise you to do the online introductory tutorial on the following website: `https://www.programiz.com/python-programming`. Note, this site advises to install python just by itself. We instead advise you to download anaconda, as this contains also the required numerical libraries.

Although coding skills are necessary in order to successfully make the assignments, they are not the main focus of this course. Therefore, we outline below which parts of the online tutorial are essential. It is certainly not a bad idea to do the entire tutorial; you will need to develop your programming skills anyway (in your studies now but also, with very high probability, in your later professional life). However, for the assignments, the parts outlined below should be sufficient.

Essential topics in the tutorial:

[1] Some of you might wonder why we use python rather than R. There are a few reasons for this. Python is more or less the third most used programming language, after C++ and java. It is widely used by companies, while R is a niche language and hardly used outside academia. Programming OR applications is easier in python; it will also be used in other courses. In general, Python is very easy to learn. Finally, if you are interested in machine learning and artificial intelligence, python is, hands-down, the best choice.

[2] Typing yourself forces you to read the code well.

[3] There are also python environments available on the web, such as repl.it., but that is typically a bit less practical than running the code on your own machine.

- INTRODUCTION: completely
- FLOW CONTROL: completely
- FUNCTIONS:
  - Python Function
  - Function argument
  - Python Global, Local and Nonlocal
  - Python Modules
  - Python Package
- DATATYPES: everything except for Python Nested Dictionary
- FILE HANDLING: nothing
- OBJECT & CLASS:
  - Python Class (For assignment 4: *Simulation of the G/G/1 queue in continuous time*)

We will use the following libraries of python a lot:

- `numpy` provides an enormous amount of functions to handle large (multi-dimensional) arrays with numbers.
- `scipy` contains numerical recipes, such as solvers for optimization software, solvers for differential equations. `scipy.stats` contains many probability distributions and numerical methods to operate on these functions.
- `matplotlib` provides plotting functionality.

We expect you to use google to search for relevant documentation of `numpy` and so on.

A couple of remarks regarding the use of Python on your own laptop:

- Please install Python through the *Anaconda* package (website: `https://www.anaconda.com/distribution`), since this comes with all the necessary documentation. Select your operating system, click "Download" under "Python 3.7 version" and follow the steps.
- After installing, use *Spyder* to work with Python. Spyder is a graphical user interface that allows you to write and compile Python code. It is similar to RStudio (which allows you to work with the language R) and TexStudio (which allows you to work with the language LaTeX).
- You might have issues with making plots That is, you tried something like

  ```
  import matplotlib.pyplot as plt

  x = range(0, 10)
  y = range(0, 10)
  plt.plot(x, y)
  plt.show()
  ```

  but nothing happens. This issue can be solved by the following steps

  1. Restart the kernel (i.e. the thing/screen in which your output is presented. In Windows you can click on a small red cross. In macOS you need to click on the options symbol and select "restart kernel")
  2. Remove the line `matplotlib.use('pdf')`, or comment it by putting a `#` symbol the start of the line.
  3. Now it should work!

1 TUTORIAL 1: EXPONENTIAL DISTRIBUTION

The aim of this tutorial is to empirically show a fascinating fact: even for very small populations in which individuals decide independently to visit a server (a shop, a hospital, etc), the exponential distribution is a good model for the inter-arrival times as seen by the server. We will develop a simulation to motivate this 'fact of nature'. In particular, our aim is to build the analogues of Figs. 1 to 3 in terms of cdfs instead of pdfs.

I expect that you will be able to cover up to about Section 1.5 during the tutorial. The rest of the sections are homework. All solutions are at the end of this document. I also make the code available on github in the form of one complete Python program and as a Jupyter notebook.

If you want to increase your probabilistic intuition, coding skills, and knowledge of data analysis, try to do all exercises by yourself. This is quite challenging, if you get stuck, check the solutions. Otherwise, if this is too much of challenge, just start from the solutions or the Jupyter notebook.

1.1 *Background*

We discuss an example to intuitively see how the exponential distribution originates. Consider a group of N patients that have to visit a hospital for regular checkup, for instance, after each visit they have to see an MD somewhere between 4 to 6 weeks later. Let us assume that the inter-arrival times $\{X_k^i, k = 1,2,\ldots\}$ of patient i are reasonable well modeled as uniform distributed between 4 and 6 weeks, i.e., $X_k^i \sim U[4,6]$. Then, with $A_0^i = 0$ for all i, define

$$A_k^i = A_{k-1}^i + X_k^i = \sum_{j=1}^{k} X_j^i \tag{1.1}$$

as the arrival moment of the kth visit of patient i.

Now the hospital doctor 'sees' the superposition of the arrivals of all patients. One way to compute the arrival moments of all patients together is to put all the arrival times $\{A_k^i, k = 1,\ldots,n, i = 1,\ldots,N\}$ into one set, and sort these numbers in increasing order. This results in the (sorted) set of all arrival times $\{A_k, k = 1,2,\ldots\}$ as seen by the doctor. Taking $A_0 = 0$, it follows hat

$$X_k = A_k - A_{k-1}, \tag{1.2}$$

is the inter-arrival time between the $k-1$th and kth patient as seen by the doctor. Thus, with this procedure, starting from inter-arrival times of individual patients, we can construct inter-arrival times of the entire population as seen by the doctor.

To plot the empirical distribution function, of $\{X_k\}$, we just count the number of inter-arrival times smaller than time t for any t. In other words, we compute the empirical distribution of $\{X_k\}$ as

$$\mathsf{P}_n(X \le t) = \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{X_k \le t},$$

where the *indicator function* is $\mathbb{1}_{X_k \le t} = 1$ if $X_k \le t$ and $\mathbb{1}_{X_k \le t} = 0$ if $X_k > t$.

In the three panels in Figure 1 we show the emperical distribution for three different cases. In the first example, we take $N = 1$ patient and let the computer generate $n = 100$ uniformly distributed numbers on the set $[4,6]$. Thus, the time between two visits of this single patient is somewhere between 4 and 6 hours, and the average inter-arrival times $\mathsf{E}[X] = 5$. For the second simulation we take $n = 100$ visits for $N = 3$ patients, and in the third, $N = 10$ patients. We add as a reference (the continuous curve) the density of the exponential distribution $\lambda e^{-\lambda t}$ for $\lambda = 1/5$, $\lambda = 3/5$ and $\lambda = 10/5$, respectively. (Recall that when one person visits the doctor with an average inter-arrival time of 5 hours, the arrival rate is 1/5. Hence, when N patients visit the doctor, each with an average inter-arrival time of 5 hours, the total arrival rate as seen by the doctor must be $N/5$.)

In Figure 2 we extend the number of visits to $n = 1000$, and in Figure 3 we take the inter-arrival times to be normally distributed times with mean 5 and $\sigma = 1$.

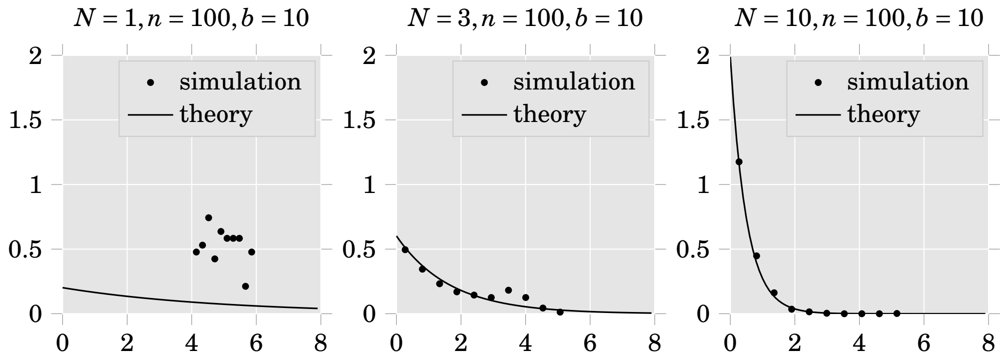

Figure 1: The inter-arrival process as seen by the doctor owner. Observe that the density $\lambda e^{-\lambda x}$ intersects the y-axis at level $N/5$, which is equal to the arrival rate when N persons visit the doctor. The parameter $n = 100$ is the simulation length, i.e., the number of visits per patient, and $b = 10$ is the number of bins to collect the data, see Remark 1.1.

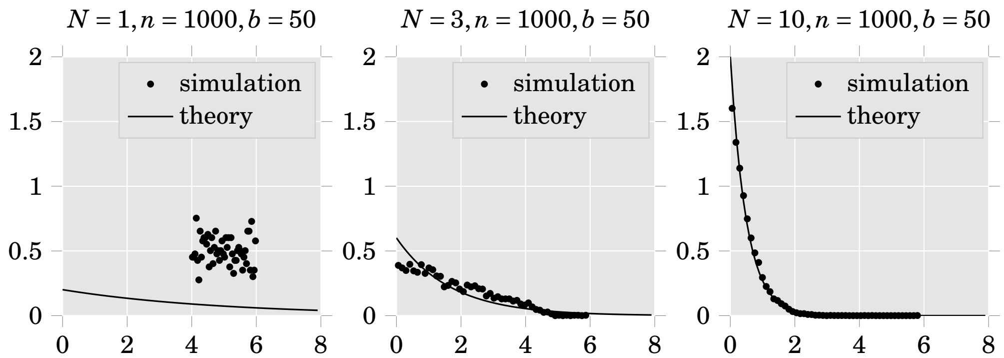

Figure 2: Each of the N patients visits the doctor at uniformly distributed inter-arrival times, but now the number of visits is $n = 1000$.

As the graphs show, even when the patient population consists of 10 members, each visiting the doctor with an inter-arrival time that is quite 'far' from exponential, the distribution of the inter-arrival times as observed by the doctor is very well approximated by an exponential distribution. Thus, it appears reasonable to use the exponential distribution to model inter-arrival times of patients for systems (such as a doctor, or a hospital or a call center) that handle many (thousands of) patients each of which deciding independently to visit the system.

Remark 1.1. The computations of the bins in Figs. 1 to 3 is a bit subtle, in particular the height. The procedure works as follows. Let $a = (X_1, \ldots, X_n)$ denote the simulated inter-arrival times. Define the width of a bin as $\delta = (\max\{a\} - \min\{a\})/b$, where b is specified by the user, for instance $b = 10$. Thus, if $b = 10$, the total range of observations is divided in $b = 10$ cells. Then the height of the ith bin is computed as $h_i = n_i/(n\delta)$ where n_i is the number of observations in the ith cell. Observe that then $\sum_{i=1}^{b} h_i \delta = 1$.

1.2 *Simulations*

1.1. Make a plan of the steps you have to carry out to make an analogue of Figure 1.1 of the queueing book in terms of a cdf. In the next set of exercises we'll carry out these steps. So please do not read on before having thought about this problem, but spend some 5 minutes to think about how to approach the problem and how to chop it up into simple steps. Then organize the steps into a logical sequence. Don't worry at first about how to convert your ideas into computer code. Coding

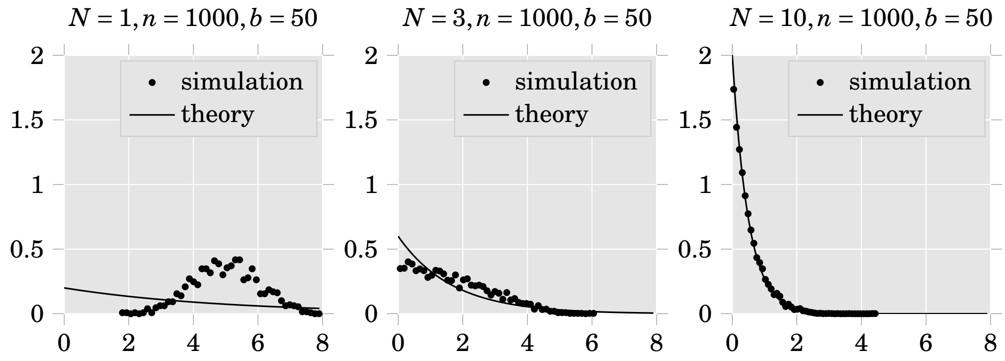

Figure 3: Each of the N patients visits the doctor with normally distributed inter-arrival times with $\mu = 5$ and $\sigma = 1$.

is a separate activity. (As a matter of fact, I always start with making a small plan on how to turn an idea into code, and I call this step 'modeling'. Typically this is a creative step, and not easy.)

We need some python libraries to make our life a bit easier. You should copy this code into your editor.

```
import numpy as np
import scipy
import matplotlib.pyplot as plt

plt.ion()  # you can skip this, I only use it for testing
           # purposes so that the computer skips making the graphs.

# this is to print not too many digits to the screen
np.set_printoptions(precision=3)

# fix the seed
scipy.random.seed(3)
```

You need to know what the seed is of random number generator; search for it on the web.

1.3 *Empirical distributions*

An important step in a simulation is to analyze the output data. For this we need the *empirical distribution*, which is defined for a set of numbers $\{a_1, \ldots, a_n\}$ as

$$F(x) = \frac{\#\{i : a_i \leq x\}}{n}. \tag{1.3}$$

We note that the computation of F is much more interesting (and challenging) than you might think[4].

Before designing an algorithm to compute the emperical distribution, it is best to first make an emperical distribution by hand, and then formalize our steps.

1.2. Suppose you are given the following sample from a population: 3.2, 4, 4, 1.3, 8.5, 9. Make the emperical distribution. Then analyze all steps you took to make the empirical distribution function.

Attempt to turn your ideas into an algorithm. You'll see that this is harder than you might have guessed before you really tried. After some thought and trying, read (and study) the code in the solution.

[4] If you search the web, you will see that computing the empirical *density* function, rather than the *distribution* fundction, is even more challenging.

1.3. The method provided by the (solution of the) previous exercise is simple, but not completely correct. What is wrong?

1.4. Take s as the sorted version of the list a. Make a plot (by hand) of s, that is, plot the points (i, s_i). Observe the crucially important fact that the function $i \to s_i$ is the inverse of the (unnormalized) distribution F.

1.5. Find now a way to invert $i \to s_i$, normalize the function to get a distribution, and make a new plot.

1.6. In the previous exercise (read the solution), we start y with 1 and end with `len(a)+1`. Why is that?

You should know that `for` loops in python are quite slow (and `for` loops in R seem to be really dramatic). For large amounts of data it is better to use `numpy`.

1.7. Use the `numpy` functions `np.arange` and `np.sort` to speed up the algorithm of the previous exercise.

With the algorithm of Exercise 1.7 we can compute and plot a distribution function of inter-arrival times specified by a list (vector, array) a. For our present goals this suffices.

If you like details, you should notice that our plot of the distribution function is still not entirely OK: the graph should make jumps, but it doesn't. Moreover, our cdf is not a real function, it can be of the form $x = (1, 1, 3)$, $y = (0, 0.5, 1)$. In the rest of this subsection we repair these points. You can skip this if you are not interested.

1.8. Read about the `drawstyle` option of the `plot` function of `matplotlib` to see how to make jumps in plots.

1.9. But now we still have vertical lines. To remove those, check how to use `hlines`.

1.10. Finally, we can make the computation of the cdf significantly faster with using the following `numpy` functions.

1. `numpy.unique`
2. `numpy.sort`
3. `numpy.cumsum`
4. `numpy.sum`

How can you use these to compute the cdf?

1.4 *Simulating the arrival process of a single patient*

With the above work we have the tools to compute the distribution function of a number of measurements or observations of some (stochastic) process. In queueing theory we are mostly interested in inter-arrival times and service times of customers. In this section we focus on the inter-arrival times of recurrent visits of a single patient at a doctor, in weeks say.

We first use `scipy` to generate sequences of i.i.d. random numbers with a given uniform distribution. Then we compute the emperical cdf for these data and compare it graphically the cdf of the uniform and the exponential distribution. A more formal method to compare cdfs is by means of the Kolmogorov-Smirnov statistic (see Wikipedia); we develop this concept in passing.

1.11. Read the documentation of the `uniform` class of the `scipy.stats` module[5]. Check in particular the documentation of the `rvs()` function. Use the documentation to see how to Generate 3 random numbers uniformly distributed on $[4, 6]$. Print these numbers to check whether you get something decent.

[5] When coding you should develop the habit to look up things on the web

1.12. Generate $L = 300$ random numbers $\sim U[4,6]$ and make a histogram of these numbers. For this, you can use the `hist` function of `matplotlib`; see the web for the documentation, in particular the `density` and `cumulative` options of the `hist` function are useful for our purposes. Then compute the empirical distribution function of these inter-arrival times, and plot the cdf.

1.13. We would like to numerically compare the empirical distribution of the inter-arrival times to the theoretical distribution, which is uniform in this case. For this we can use the Kolmogorov-Smirnov statistic. Try to come up with a method to compute this statistic, and then compute it. Look up a table with critical values for the Kolmogorov-Smirnov test statistic. Given a 5% confidence level, do we reject the hypothesis that our sample is drawn from a $U[4,6]$ distribution?

You might find the following functions helpful (read the documentation on the web to see what they do):

1. `numpy.max`
2. `numpy.abs`
3. `scipy.stats.uniform`, the `cdf` function.

1.14. Finally, plot the empirical distribution and the exponential distribution with mean $\lambda = 1/5$ in one graph. (Why do we take $\lambda = 1/5$?) The relevant `scipy` functions can be found with searching on `scipy.stats.expon`.

Explain why these graphs are different. In passing, compute the KS statistics to compare the simulated inter-arrival times with an exponential distribution. Do we reject the hypothesis that our sample is drawn from this exponential distribution?

1.5 *Simulating many patients*

We would now like to simulate the inter-arrival process as seen by a doctor that serves many recurring patients. For ease, we call this the *merged*, or *superposed*, inter-arrival process.

Making the merged process out of a number of individual patient arrival process turns out to an interesting algorithmic challenge. Thus, we start with a numerical example with two patients, and organize all steps we make to compute the empirical distribution of the merged inter-arrival process. Then we make an algorithm, and scale up to arbitrary numbers of patients.

1.15. Suppose we have two patients with inter-arrival times $a = [4,3,1.2,5]$ and $b = [2,0.5,9]$. Make by hand the empirical cdf of the merged process. Then summarize the steps you took in the process.

1.16. The steps of the previous exercise can be summarized by the following code:

```
from itertools import chain

def superposition(a):
    A = np.cumsum(a, axis=1)
    A = list(sorted(chain.from_iterable(A)))
    return np.diff(A)
```

Note that the input `a` in the function `superposition` is a matrix of inter-arrival times of several patients.

Study this code by reading the documentation (on the web) of the following functions, `numpy.cumsum`, in particular read about the meaning of axis, `itertools.chain.from_iterable`, and `numpy.diff`.

1.17. Generate 100 random inter-arrival times $\sim U[4,6]$ for 3 individual patients. Then use the code above to compute the inter-arrival times of the merged stream. Plot the emperical cdf of the merged inter-arrival times, and compare the emperical cdf to the exponential distribution with the correct mean (which should by $\lambda = 3/5$). Also compute the Kolmogorov-Smirnov statistic for this case.

What is the effect of increasing the number of patients from $N = 1$ to $N = 3$?

1.18. Compare the empirical distribution of the inter-arrival times generated by $N = 10$ patients to the exponential distribution (What is now the appropriate arrival rate:). Make a plot, and explain what you see.

1.19. Do the same for $N = 10$ patients with normally distributed inter-arrival times with $\mu = 5$, and $\sigma = 1$. For this use `scipy.stats.norm`. What do you see? What is the influence of the distribution of inter-arrival times of an individual patient?

1.20. If $\mu = \sigma = 5$ then the analysis should break down. What happens if you try this settting? If you don't get real strange results, the code itself must be wrong[6].

1.6 *Memoryless property*

It is well known that the exponential distribution has the memoryless property, that is, when X has an exponential distribution,

$$\mathsf{P}(X > s + t \,|\, X > s) = \mathsf{P}(X > t),$$

for $s, t \geq 0$. Can we see this property in the data obtained by simulation?

1.21. There is a nice `numpy` method to select (or filter) data that satisfies a certain property. Here, to select all inter-arrival times in the list `a` that are larger than some s, we can use the code

```
a = a[a>s]
```

With this, modify one of the above functions in which we compared the emperical distribution of the data to the theoretical distribution. Take $s = 0.5$.

1.22. As is apparent from the previous exercise, for small s, such as $s = 0.5$, the data satisfies the memory. But what happens when the threshold s becomes larger, e.g., 2?

1.23. When $s = 2$ the memoryless property does not seem to hold. Suppose we take more data points, in other words, let's take $N = 10^5$, and see what happens.

1.24. What can you conclude from this analysis?

1.7 *The number arrivals in an interval and the Poisson distribution*

Besides the memory-less property, we can also analyze whether our inter-arrival times satisfy the 'Poisson property'. That is, when the inter-arrival times are exponential with arrival rate λ, the number of observations in an interval of a fixed length t should be Poisson distributed with mean λt. In this final section we will try to see whether this 'Poisson' property holds for our patient inter-arrival times.

1.25. With the code below we can compute the number of arrivals per interval for the list `A` that contains the arrival times of patients. Then we compare the emperical distribution of the number of arrivals in an interval and compare it to the Poisson distribution with the same mean. Explain the code, and then run it.

```python
def poisson_1():
    N, L = 10, 1000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    # G = norm(loc=5, scale=1)
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2
```

[6] When testing code it is also a good idea to see what happens if you use bogus numbers. The program should fail or give very strange results.

```
    mean = 20  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
    # plt.pause(10)
    plt.show()

poisson_1()
```

1.26. Run the code for various choices of `mean`, for instance, take `mean=5`, and then compare this to a case with `mean=20`. You should notice that when `mean` is large, 20 or so, the variance of the number of arrivals is quite a bit smaller than the variance of the Poisson distribution. Thus, eventually, it becomes clear that the inter-arrival times are more regular than exponential. All in all, we see that data analysis is not simple.

1.27. As a final check on our implementation, we can generate exponentially distributed inter-arrival times rather than the uniformly distributed ones we have been using all the time. Implement this idea, and check the results.

1.8 *Summary*

1.28. Make a summary of what you have learned from this tutorial.

2 TUTORIAL 2: SIMULATION OF THE $G/G/1$ QUEUE IN DISCRETE TIME

In this tutorial we simulate the queueing behavior of a supermarket or hospital, but it is easy to extend the ideas to much more general queueing system. We first make a simple model of a queueing system, and then extend this to cover more difficult queueing situations. With these models we can provide insight into how to design or improve real-world queueing systems. You will see, hopefully, how astonishingly easy it is to evaluate many types of decisions and design problems with simulation.

For ease we consider a queueing system in discrete time, and don't make a distinction between the number of jobs in the system and the number of jobs in queue. Thus, queue length corresponds here to all jobs in the system, cf. Section 1.4 of the queueing book.

2.1 *Set up*

2.1. Write down the recursions to compute the queue length at the end of a period based on the number of arrivals a_i during period i, the queue length Q_0 at the start, and the number of services s_i. Assume that service is provided at the start of the period. Then sketch an algorithm (in pseudo code) to carry out the computations (use a for loop). Then check this exercise's solution for the python code.

With the code of the above exercises we can start our experiments.

2.2. Copy the code of the previous exercise to a new file in Anaconda. Then add the code below and run it. (Or use the jupyter notebook.)

Here λ is the arrival rate, μ the service rate, N the number of periods, and q_0 the starting level of the queue. Explain what the code does. Can you also explain the value of the mean and the standard deviation?

```python
def experiment_1():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    print(a.mean(), a.std())

experiment_1()
```

2.3. Modify the appropriate parts of code of the previous exercise to the below, and run it. Explain what you see.

```python
def experiment_2():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)

    plt.plot(Q)
    plt.show()
    print(d.mean())


experiment_2()
```

2.4. Getting statistics is really easy now. For example, try to plot the empirical distribution function of the queue length process.

2.5. Can you explain the value of the mean number of departures?

2.6. Plot the queue length process for a large initial queue, for instance, with

```
def experiment_4():
    labda, mu = 5, 6
    q0, N = 1000, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)
    plt.plot(Q)
    plt.show()


experiment_4()
```

Explain what you see.

2.7. Set $q_0 = 10000$ and $N = 1000$. (In Anaconda you can just change the numbers and run the code again, in other words, you don't have to copy all the code.) Finally, make these values again 10 times larger.

Explain what you see. What is the drain rate of Q?

2.8. What do you expect to see when $\lambda = 6$ and $\mu = 5$? Once you have formulated your hypothesis, check it.

2.2 *What-if analysis*

With simulation we can do all kinds of experiments to see whether the performance of a queueing system improves in the way we want. Here is a simple example of the type of experiments we can run now.

For instance, the mean and the sigma of the queue length might be too large, that is, customers complain about long waiting times. Suppose we are able, with significant technological investments, to make the service times more predictable. Then we would like to know the influence of this change on the queue length.

To quantify the effect of regularity of service times we first assume that the service times are exponentially distributed; then we change it to deterministic times. As deterministic service times are the best we can achieve, we cannot do any better than this by just making the service times more regular. If we are still unhappy about the effect of making service times much more regular, we have to make the average service times shorter, or add extra servers, or block demand so that the inflow reduces.

2.9. Let's test the influence of service time variability. Replace the service times with the code `s = np.ones_like(a) * mu`. Read the docs of `numpy` to see what `np.ones_like` does. Explain the result.

2.10. Suppose that, instead of being able to reduce the variability of the service times, we are able to reduce the average service time by 10%, say. To see the effect of this change, replace `s` by `s = poisson(1.1*mu).rvs(N)`, i.e., we increase the service rate with 10%.

Do the computations and compare the results to those of the previous exercise. What change in average service time do we need to get about the same average queue length as the one for the queueing system with deterministic service times? Is an 10% increase enough, or should it be 20%, or 30%? Just test a few numbers and see what you get.

I hope you make the crucial observation from the above cases that we can experiment with all kinds of system changes and compare their effects on, for instance, waiting times. In fact, with simulation we can easily carry out 'what-if' analysis.

2.3 *Control*

In the previous section we analyzed the effect of the design of the system, such as changing the average service times. These changes are independent of the queue length. In many systems, however, the service rate depends on the dynamics of the queue process. When the queue is large, service rates increase; when the queue is small, service rates decrease. This type of policy can often be seen in supermarkets: extra cashiers will open when the queue length increases, and some of them close when all queues are empty.

Suppose that normally we have 6 servers, each working at a rate of 1 customer per period. When the queue becomes longer than 20 (this is about 3 customers per server in queue) we install two extra servers, and when the queue is empty again, we switch off the extra two servers, until the queue hits 20 again, and so on.

2.11. Try to write pseudo-code (or a python program) to simulate the queue process. (This is pretty challenging; it's not a problem if you spend quite some time on this.)

Another way to deal with large queues is to simply block customers if the queue is too long. What would happen if we would customers to enter the system when the queue is 15 or larger? How, for instance, would that affect the average queue and the distribution of the queue?

2.12. Modify the code to include blocking and do an experiment to see the effect on the queue length.

2.4 *Cost computations*

Let us next consider a single server queue that can be switched on and off. There is a cost h associated with keeping a job waiting for one period, there is a cost p to hire the server for one period, and it costs s to switch on the server. Given the parameter values of the next exercise, what would be a good threshold θ such that the server is switched on when the queue hits or exceeds θ? We assume (and it is easy to prove) that it is optimal to switch off the server when the queue is empty.

2.13. Jobs arrive at rate $\lambda = 0.3$ per period. If the server is present the service rate is $\mu = 1$ per period; if the server is 'on vacation', the service rate is 0. The number of arrivals and services per period are Poisson distributed with the given rates. Then, $h = 1$ (without loss of generality), $p = 5$ and $S = 500$. Write a simulator to compute the average cost for setting $\theta = 100$. Then, change N to find a better value.

As a final example, suppose we have to pay a penalty for any customer that sees a queue larger than 20. How to estimate the effect of these costs?

2.14. Suppose that the number of jobs arriving in a period is Poisson distributed with $\lambda = 5$, and the number of potential services is $\mu = 6$ per period. Use the programs we developed earlier to compute the queue lengths, and use the code `(Q>20)` to count all periods in in which $Q > 20$.

2.5 *Summary*

2.15. What have you learned from this tutorial? What interesting extensions can you think of, in particular, what is relevant for practice?

3 TUTORIAL 3: SIMULATION OF THE $G/G/1$ QUEUE IN CONTINUOUS TIME

In this tutorial we will write a simulator for the $G/G/1$ queue. For this we use two concepts that are essential to simulate any stochastic system of reasonable complexity: an *event stack* (or event schedule) to keep track of the sequence in which events occur, and *classes* to organize data and behavior into single logical units. We will work in steps towards our goal; along the way you will learn a number of fundamental and highly interesting concepts such as *classes* and efficient *data structures*. Once you have understood the implementation of the $G/G/1$ queue that we are going to build in this tutorial, discrete event simulators are no longer a black box for you. Hence, what you will learn from this tutorial extends well beyond the simulation of the $G/G/1$ queueing process.

Once we have built our simulator we use it to analyze the queueing behavior at a check-in desk at an airport.

In Section 4 we will generalize the simulator to the $G/G/c$ queue and clean up the code by working with a class for the $G/G/c$ queue. We can then also extend the case accordingly.

3.1 *Sorting with event stacks*

If we simulate a stochastic system we like to move from event to event. To illustrate this idea, consider a queueing process such as the $M/M/1$ queue. It is clear that only at arrival and departure epochs something interesting happens, so that we can neglect any time in between arrival and departure epochs as nothing happens. Hence, in our simulator we prefer not to keep track of the entire time line, but just of the relevant epochs. Once we have mechanism to store and organize these events, we can jump from event to event.

Clearly, we want to follow the sequence of events in the correct order of time. For this we use an *event stack*. To see how this works, we will consider a simple example first, and then we extend to more difficult situations.

Suppose we have 4 students, and we like to sort them in increasing order of age. One way to do this is to insert them into a list such that the insertion process respects the correct ordering right away. For this very general problem (insert elements in an ordered way in a list) a number of efficient data structures has been developed, and one of these is the *heap queue*.

3.1. Search the web on `python` and `heap queue`. Study the examples and write some code to sort the students Jan, Pete, Clair, and Cynthia, with ages 21, 20, 18, and 25, in order of increasing age.

3.2. Extend the code of the previous exercise such that we can include more information with the ages than just the name, for instance, also the brand of their mobile phone.

3.3. It may seem that we have just solved a simple toy problem, but this is not true. In fact, we have established something of real importance: heap queues form the core functionality of (nearly) all discrete event simulators used around the world. To help you understand this fact, develop some ideas how you could to use heap queues to simulate a queueing process.

3.2 *G/G/1 queue*

In this section we will set up a simulator for the $G/G/1$ queue with an event stack. Let us work in steps toward the final simulator.

We start with some basic imports, initialize the stack, and define two numbers `ARRIVAL` and `DEPARTURE` that will be used to tag the type of event that will be popped from the event stack. So, make a new file, and type the following at the top of it.

```
ARRIVAL = 0
DEPARTURE = 1

stack = [] # this is the event stack
```

3.4. Why do we introduce the variables `ARRIVAL` and `DEPARTURE`; we could also use the numbers 0 and 1, and this works just as well?

First of all we need jobs with arrival times and service times. The easiest way to handle jobs in a simulator is by means of a class, as in the code below.

```
class Job:
    def __init__(self):
        self.arrival_time = 0
        self.service_time = 0
        self.departure_time = 0
        self.queue_length_at_arrival = 0

    def sojourn_time(self):
        return self.departure_time - self.arrival_time

    def waiting_time(self):
        return self.sojourn_time() - self.service_time

    def __repr__(self):
        return f"{self.arrival_time}, {self.service_time}, {self.departure_time}\n"

    def __le__(self, other):
        # this is necessary to sort jobs when they have the same arrival times.
        return self.id <= other.id
```

A class has a number of *attributes*, such as `self.arrival_time`, to capture its state and a number of functions, such as `def waiting_time(self)`, to compute specific information that relates to the class[7]. We initialize the arrival time and service time to zero, and we we use the attributes `departure_time` and `queue_length_at_arrival` for a statistical analysis at the end of the simulation. Finally, we have functions to compute the waiting time and the sojourn time, and `__repr__` to print info about the job. (We also need the function `__le__` for a technical reason, which you do not need to know if you are a python newbie. When two jobs have the same arrival time, the heap queue needs extra information to sort such jobs. For this we simply use `id` of the object.)

Note that our naming of functions also acts as documentation of what the functions do. Note also that member variables and functions in python start with the word `self`; this is to distinguish the (value of the) member variables from variables with the same name but lying outside the scope of the class.

Classes are extremely useful programming concepts, as it enables you to organize state (that is, attributes) and behavior (that is, functions that apply to the attributes) into logical components. Moreover, classes can offer functionality to a programmer without the programmer needing to understand how this functionality is built. Classes offer many more advantages such as inheritance, but we will not discuss that here.

Once we have a class, we can make an object with the code `job = Job()`.

3.5. Check the internet to find out the difference between a 'class' and an 'object'.

3.6. Make 10 jobs with exponentially distributed inter-arrival times, with $\lambda = 2$, and exponentially distributed service times with $\mu = 3$. Put these jobs on an event stack, and print them in order of arrival time. Tag the events with the job and event type (which is an arrival).

Now that we know how to make jobs and put them on an event stack, we can make a plan to simulate the $G/G/1$ queue.

Replace the while loop of the previous exercise by the code below. Observe that we split events into two types[8]: arrivals and departures.

[7] In python, functions belonging to a class are called 'methods' or 'member functions'.
[8] The word `type` is a reserved word in python, hence I use `typ` instead.

```
while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
```

3.7. With the above idea to make a while loop, make a list of things that have to take place at an arrival event (in particular, what should happen when the server is busy at an arrival epoch, what should happen if the server is idle?) and at a departure event (in particular, what if the queue is empty, or not empty)?

Turn your ideas into code. It's not a problem if you spend some time on this; you will learn a lot in the process.

3.8. Once you have all your code (or you understand the code in the solutions) run an experiment with 10 jobs with exponentially distributed inter-arrival times with $\lambda = 2$ and service times $\sim Exp(\mu)$ with $\mu = 3$.

3.3 *Simple tests*

As a general observation, testing code is very important. For this reason, before we can apply our simulator to real situations, we should apply it to some simple cases that we can analyze with theory. Of course the results of the simulation and the models should coincide.

3.9. Run a simulation for the $M/M/1$ queue with $\lambda = 2$ and $\mu = 3$ with 100 jobs and compute the average queue length. Then extend to 1000, and then to 10^5. Compare it to the theoretical expected queue length of the $M/M/1$ queue at arrival moments. Do the same thing for the average sojourn time.

3.10. Thus, let us get further confidence in our $G/G/1$ simulator by specializing it to the $D/D/1$ queue. Check the documentation of the `uniform` distribution in `scipy.stats` to see what the following code does.

```
from scipy.stats import uniform

stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def experiment_3():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = uniform(3, 0.00001)
    G = uniform(2, 0.00001)
    num_jobs = 10

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))
```

```
    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    for j in served_jobs:
        print(j)


experiment_3()
```

Then use this in our simulation. What happens? What happens if you reverse the 2 and the 3 in F and G?

3.4 *Comparison with the $M/G/1$ queue*

As a second set of tests, we can compare the results of our simulator with those obtained for the $M/G/1$ queue. For this queue, we have a closed-form expression for the average waiting time and an algorithm to compute the queue-length distribution. So, we can, and should, use these theoretical results as further tests.

Assume that demand arrives as a Poisson process with rate $\lambda = 1/3$ per minute, and that the service distribution is $U[1,3]$ minute.

3.11. Implement the Pollaczek-Khinchine equation in python and use this to compute the expected queue length.

3.12. Estimate the average queue length with the our $G/G/1$ simulator and compare the results with the theoretical result obtained from the PK-formala. Note that this is another test of our implementation.

3.5 *The check-in process at an airport*

Now that we have a simulator—and we tested it, although not sufficiently well to use it for real applications—we can apply it to analyze the queueing behavior at a check-in process at an airport. For ease we focus here on a single server desk, in the next tutorials we extend the simulator to more realistic scenarios.

We need to model the arrival process and the service times. With regard to the arrivals, we know that a plane has a certain capacity. Suppose that we deal with a small plane with just 60 seats, and that the 60 customers arrive uniformly distributed over 2 hours. The service time distribution is $\sim U[1,3]$ minute.

3.13. How can we simulate such an arrival process?

3.14. Estimate the average queue length with our $G/G/1$ simulator. Use the `Counter` class to count the queue length distribution, and comment on the result.

3.15. In my simulation output 2 customers see a queue length of 15. That is quite large. Lets plot the queue length at arrival moments to see whether that explains our results. This is the code you can use for this.

```
import matplotlib.pyplot as plt

x = [job.arrival_time for job in served_jobs]
```

```
y = [job.queue_length_at_arrival for job in served_jobs]

plt.plot(x, y, "o")
plt.show()
```

What do you see?

3.16. Change the service time distribution to $\sim U[1,2]$ to see whether that reduces the queue lengths. Suppose also that we open the desk a bit longer, half an hour say. What is the result of these changes?

3.6 *Extensions*

Here are some final, more general, questions to deepen your understanding of queues systems and simulation.

3.17. Up to now we studied the $G/G/1$ FIFO (First In First Out) queue. How to change the code to simulate a LIFO (Last In First Out) queue?

3.18. In a priority queue jobs belong to a priority class. Jobs with higher priority are served before jobs with lower priority, and within one priority class jobs are served in FIFO sequence. A concrete example of a priority queue is the check-in process of business and economy class customers at airports. How would you build a $G/G/1$ priority queue?

3.19. We implemented the queue as a python list. Why is a deque[9] a more efficient data structure to simulate a queue? Why is it important to know the efficiency of the data structures and algorithms you use?

3.20. Recall that in Exercise 3.6 we built all jobs before the queueing simulation starts. Why is this not a good decision? Note that in the simulation of $G/G/c$ queue below we will generate jobs only when needed.

3.7 *Summary*

3.21. What have you learned in this tutorial?

3.22. Can you make the simulator more realistic?

[9] Search the web to understand what this is.

4 TUTORIAL 4: $G/G/c$ QUEUE IN CONTINUOUS TIME

The goal of this tutorial is to generalize the simulator for the $G/G/1$ to a $G/G/c$ queue. You will see that this is rather easy, once you have the right idea. We will organize the entire simulator into a single class so that our code is organized and we will develop a method to set up structured tests[10] Once we have a working simulator we apply it to the check-in process of the airport.

4.1 *The simulator*

4.1. In the $G/G/1$ we have just one server which is busy or not. Generalize the `handle_arrival` function of the $G/G/1$ queue such that it can cope with multiple servers.

4.2. Likewise, generalize the `handle_departure` function of the $G/G/1$ queue such that it can cope with multiple servers.

4.3. Can you think of some sort of property of the $G/G/c$ queue that must be true at all times? It is important to use such properties as tests while developing.

4.4. The design of the simulation of $G/G/1$ queue is not entirely satisfactory. For instance, if we want to run different experiments, we have to clean up old experiments before we can start new ones. The current design is not very practical for this. It is better to encapsulate the state and its behavior (i.e., functions) of the simulator into one class. Build a class that simulates the $G/G/c$ queue; use the ideas of the $G/G/1$ simulator.

It is not problem when you spend considerable time on this exercise. Most of the following exercises are simple numerical experiments, which should take little time. So, just try and see how far you get. All code is in the solution.

4.5. How can we instantiate the `GGc` simulator, i.e., make an object of a class? How can we run a simulation with exponential inter-arrival times with $\lambda = 2$, exponential service times with $\mu = 1$, $c = 3$, and 10 jobs?

4.2 *Testing*

4.6. Test the simulator with deterministic inter-arrival times, for instance a job that arrives every 10 minutes, and each job takes 1 hour of service. Check the departure process for $c = 1$, $c = 2$, $c = 5$, $c = 6$, and $c = 7$.

4.7. For testing purposes, implement Sakasegawa's formula. BTW, why should you use Sakasegawa's formula?

4.8. Test the code for the $M/M/1$ queue with $\lambda = 0.8$ and $\mu = 1$. Compute the mean waiting time in queue and compare this to the theoretical value. Test it first for run length $N = 100$, then for $N = 10000$. Think about how to organize your tests.

4.9. Now test it for the $M/D/1$ queue with $\lambda = 0.9$ and $S \equiv 1$.

4.10. With our simulation we can also check the quality of Sakasegawa's approximation. Try this for the $M/D/2$ queue $\lambda = 1.8$ and $S \equiv 1$.

4.3 *The check-in process at an airport*

We again analyze the check-in process at an airport, but we make it more realistic than the analysis of Section 3.5 in that we now allow for multiple check-in desks.

[10] Structured testing is enormously important when you use code for real applications. (I suppose you prefer all code used in self-driving cars to be tested.) If you are interested, check the web on `python unittest` and 'test-driven coding'. With test-driven coding you first write down a hierarchy of many simple tests that your program is supposed to pass. Then you build your program until the first test pass. Then you implement the next step until the second test pass too, and so on.

Assume that a flight departing at 11 am consists of $N = 300$ people. For ease, assume that all N customers arrive between 9 and 10. Suppose that the check-in of a job is $U[1,3]$ distributed.

4.11. Model the arrival process.

The queueing problem is evidently to determine the required number of servers c such that the performance is acceptable.

4.12. What performance measures are relevant here?

4.13. Define the offered load as $\lambda \mathsf{E}[S]$ and the load as $\rho = \lambda \mathsf{E}[S]/c$. Is it important for the check-in desk that $\rho < 1$?

4.14. Why do we need simulation to analyze this case?

4.15. Search for a good c.

4.16. What do you think of the results? If you are not satisfied, propose and analyze an improvement.

4.17. Relate the results of these simulations to your traveling experiences.

4.4 *Summary*

4.18. What have you learned in this tutorial? Can you extend this work to more general cases?

5 TUTORIAL 5: AIRPORT CHECK-IN PROCESS

In this final tutorial we consider the case in which business and economy class customers have their own server(s). In fact, we are going to use simulation to compare different ways to organize the queueing system. One way is like this: there is one server strictly allocated to business class customers, and there are c, say, servers strictly used by the economy class customers. Another way is like this. There are still one business server and c economy servers. When the business server is free, this server takes just one economy customer in service. When, after this service, there is still no business customer, the business server serves a second economy customer. Once a business customer arrives, the business server finished the economy customer (if there is any), and then servers the business customer. We use a similar policy for serving business customers by free economy class server. The problem is of course to figure out which of these policies is best.

5.1. What KPI's would you use to compare the two policies?

As the simulation of this case is somewhat more difficult than the $G/G/c$ queue, testing is also more important. For this reason we will follow a *test-driven* procedure, which works like this. First we design a number of test cases, from very simple to a bit less simple. Then we build and test until the first test case passes. Then we build and test until also the second case passes, and so on, until all the tests pass. And, indeed, along the way we caught numerous mistakes, from missing `if` statements (a very big bug) to simple typos.

5.1 *The simulator*

5.2. Make a list of simple cases whose outcome you can check by hand, to cases that are a bit harder but for which you can use theoretical tools to validate the outcome.

Before we build the class for the queues we observe that the implementation of the $G/G/c$ of Section 4 suffers from a performance penalty. The problem is that we create time and again new objects in the calls `F.rvs()` and `G.rvs()` to generate the quasi-random inter-arrival and service times[11]. Specifically, code like this runs very slow:

```
def generate_jobs_bad_implementation(F, G, p_business, num_jobs):
    # the difference in performance is tremendous
    for n in range(num_jobs):
        job = Job()
        job.arrival_time = time + F.rvs()
        job.service_time = G.rvs()
        if uniform(0,1).rvs() < p_business:
            job.customer_type = BUSINESS
        else:
            job.customer_type = ECONOMY

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

Note that we generate `num_jobs` jobs. Then, with probability p (`p_business` in our code) the job is a business customer.

To repair this problem it is better to let `scipy` call a random number generator in C++ just once and generate all random numbers in one pass. Here is one way to obtain a speed up of about a factor 100.

[11] You only know such things if you have some general knowledge of python or computer programming general. Hence, it pays to invest in knowledge of the computer language you use.

```python
def generate_jobs(F, G, p_business, num_jobs):
    jobs = set()
    time = 0
    a = F.rvs(num_jobs)
    b = G.rvs(num_jobs)
    p = uniform(0, 1).rvs(num_jobs)

    for n in range(num_jobs):
        job = Job()
        job.arrival_time = time + a[n]
        job.service_time = b[n]
        if p[n] <  p_business:
            job.customer_type = BUSINESS
        else:
            job.customer_type = ECONOMY

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

We assume that the service distribution is the same for all job classes. If we do not want this, we can make the service time dependent on the job type.

5.3. How to implement the dependence of job service time on job type?

5.4. Try to implement (or design) the $G/G/c$ queue with business customers. The rules must be like this. Business and economy class customers have separate queues and separate, dedicated servers. There is one business server and there are c economy class servers. If the business (economy) server becomes idle and there is an economy (business) class customer in queue, the business (economy) server takes an economy (business) customer into service.

You can find the code in `tutorial_5.py`. Read it carefully so that you understand all details.

5.5. To design some further test cases: think about what happens if two customer classes are completely separated: business customers have one dedicated server and the economy customers have c dedicated servers. How would you implement this in the code?

5.6. Suppose there are no business customers, then all servers should be available for the economy class customers. With this idea you can implement the $M/M/c$ queue and check whether the simulator gives the same results as the theoretical values.

5.2 *Case analysis and summary*

Recall, we build this simulator to see whether we want to share the servers or not. To obtain insight into the performance of each situation (shared, or unshared), we need to analyze a few characteristic situations. The result of this might be that in certain settings, sharing is best (relatively many business customer?), and in others, we don't want to share.

5.7. Assume we have $n = 300$ customers, the desks open 2 hours before departure and close 1 hour before departure, and service distribution is $U[1,2]$ (minutes) for all customers. The fracton of business customers is $p = 0.1$. There are $c = 6$ servers for the economy class customers, and 1 for the business customers. Finally, assume that during the opening slot of the desks, the arrival process is Poisson with constant rate.

Make a 'test-bed' that allows you to compare, side-by-side, the shared versus the unshared situation. Then, do the simulation and interpret the results.

5.8. Now start varying on the base case, for instance, longer opening hours, more (less) customers, higher (lower) fraction of business customers, longer (shorter) service times, more (less) variability in the service times. (In other words, just play with your simulator, and try to relate the outcomes to what you experienced/observed from your last flight.) Compare the results.

5.9. What remains to be done so that you can apply our simulator to a real check-in process? In other words, what do you have to do to use this tool in a real consultancy job?

HINTS

h.1.15. Note that the doctor sees the combined arrival process, so first find a way to merge the arrival times of the patients into one arrival process as observed by the doctor. Then convert this into inter-arrival times at the doctor.

h.2.4. For the empirical distribution function you can use the code of Exercise 1.7. Before looking it up, try to recall how the cdf is computed.

h.2.11. As a hint, you need a state variable to track whether the extra servers are present or not. The solution shows the code.) Analyze the effect of the threshold at 20; what happens if you set it to 18, say, or 30? What is the effect of the number of extra servers; what happens if you would add 3 instead of 2?

h.3.9. You might want to check Exercise 3.6 to see how to generate the jobs.

h.3.11. Build a function with arguments `labda` and `G`, where `G` represents a distribution. Next use `G.var()` and `G.mean()` to compute c_e^2.

h.3.12. Reset all data and clear all lists.

h.3.14. Reset all data and clear all lists.

h.3.18. We have used a good data structure already. How should this be applied?

h.3.19. Search the web on `python` and `deque`.

h.4.1. Introduce a `num_busy` variable that keeps track of the number of busy servers. What happens if a job arrives and this number is less than c; what if this number is equal to c?

h.4.2. Again, use the `num_busy` variable.

h.4.3. There must be a relation between the queue length and the number of busy servers.

h.4.7. Follow the implementation of Exercise 3.11.

h.4.8. Implement the test in a function so that you can organize all your setting in one clean environment. We can use this below for more general cases.

h.4.9. Implement the deterministic distribution as `uniform(1, 0.00001)`.

h.4.11. Most people arrive independently. What is the arrival rate?

h.4.13. Think hard about the context. Is this queueing process stationary? Is the arrival process stationary?

h.4.15. For this it is best to write two functions. The first function simulates a scenario for a fixed set of parameters and computes the performance measures. The second function calls the simulator over a suitable range for c and prints the KPIs. Like this, we use the second function to fix all parameters so that we always can retrieve the exact scenarios we analyzed.

h.5.2. Suppose there is only a server for business customers and also only business customers? What should happen? Suppose further that the inter-arrival times are constant and 1 minutes and service times constant and equal to 0.5 minute? What if the service times are 2 minutes and 10 business customers arrive? What if all 10 customers arrive at time 0; when should the last leave? What if we only have economy customers but only the business server? And so on. Ensure you design cases you can check by hand. These simple tests will catch many, many errors. As a general rule, invest in such tests.

SOLUTIONS

s.1.1.

1. Generate realizations of a uniformly distributed random variable representing the inter-arrival times of one patient.
2. Plot the inter-arrival times.
3. Compute the (empirical) distribution function of the simulated inter-arrival times.
4. Plot the (empirical) distribution function.
5. Generate realizations of uniformly distributed random variables representing the inter-arrival times of multiple patients, e.g., 3.
6. Compute the arrival times for each patient.
7. Merge the arrival times for all patients. This is the arrival process as seen by the doctor.
8. Compute the inter-arrival times as seen by the doctor.
9. Plot these inter-arrival times.
10. Compare to the exponential distribution function with a suitable arrival rate λ.

s.1.2. We put the algorithm in a function so that we can use it later. The algorithm is useful to study, but it has some weak points. In the exercises below we will repair the problems.

We run a test at the end.

```python
def cdf_simple(a):
    a = sorted(a)

    # We need the support of the distribution. For this, we need
    # to start slightly to the left of the smallest value of a,
    # and stop somewhat to the right of the largest value of a. This is
    # realized by defining m and M like so:
    m, M = int(min(a)), int(max(a)) + 1
    # Since we know that a is sorted, this next line
    # would be better, but less clear perhaps:
    # m, M = int(a[0]), int(a[-1])+1

    F = dict()  # store the function i -> F[i]
    F[m - 1] = 0  # since F[x] = 0 for all x < m
    i = 0
    for x in range(m, M):
        F[x] = F[x - 1]
        while i < len(a) and a[i] <= x:
            F[x] += 1
            i += 1

    # normalize
    for x in F.keys():
        F[x] /= len(a)

    return F

def test1():
    a = [3.2, 4, 4, 1.3, 8.5, 9]
```

```
    F = cdf_simple(a)
    print(F)
    I = range(0, len(a))
    s = sorted(a)
    plt.plot(I, s)
    plt.show()

"""
You can run a separate test, such as test1 by uncommenting the line
below, i.e., remove the # at the start of the line and the space, so
that the line starts with the word "test_1()". Once the test runs, you
can comment it again, and move to the next test. Uncomment that line,
run the program, comment it again, etc.
"""

test1()
```

s.1.3. We have to guess the support of F (the set of points where F makes the jumps) upfront, and we concentrated the support of F on the integers. However, in general F can make jumps at any real number, for instance at 3.2.

```
    # Include this void code to keep the numbering the same between
    # the tutorial text and the jupyter notebook.
```

s.1.4. In the answer we let the computer do all the work.

```
def test1a():
    a = [3.2, 4, 4, 1.3, 8.5, 9]
    I = range(0, len(a))
    s = sorted(a)
    plt.plot(I, s)
    plt.show()

test1a()
```

s.1.5. Here is one way.

```
def cdf_better(a):
    n = len(a)
    y = range(1, n + 1)
    y = [z / n for z in y]   # normalize
    x = sorted(a)
    return x, y


def test2():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf_better(a)

    plt.plot(x, y)
    plt.show()

test2()
```

s.1.6. The reason is that at s_1 the first observation occurs. Hence, the unnormalized F should make a jump of at least one at s_1. Next, the `range` function works up to, but not including, its second argument. Hence (in code), `range(10)[-1]/10 = 0.9`, that is, the last element `range(10)[-1]` of the set of numbers $0, 1, \ldots, 9$ is not 10. Hence, when we extend the range to `len(a)+1` we have a range up to and including the element we want to include.

```
# Include this void code to keep the numbering the same between the tutorial
# text and the jupyter notebook.
```

s.1.7. The following code is much, much faster, and also very clean. Note that we normalize y right away.

```
def cdf(a):  # the implementation we will use mostly. It is simple and fast.
    y = np.arange(1, len(a) + 1) / len(a)
    x = np.sort(a)
    return x, y
```

s.1.8. With the `drawstyle` option:

```
def make_nice_plots_1():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf(a)

    plt.plot(x, y, drawstyle="steps-post")
    plt.show()

make_nice_plots_1()
```

s.1.9. This is better.

```
def make_nice_plots_2():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf(a)

    left = np.concatenate(([x[0] - 1], x))
    right = np.concatenate((x, [x[-1] + 1]))

    plt.hlines(y, left, right)
    plt.show()


make_nice_plots_2()
```

There we are!.

s.1.10. Here it is.

```
def cdf_fastest(X):
    # remove multiple occurences of the same value
    unique, count = np.unique(np.sort(X), return_counts=True)
    x = unique
    y = count.cumsum() / count.sum()
    return x, y
```

s.1.11. Copy this code and run it.

```
from scipy.stats import uniform

# fix the seed
scipy.random.seed(3)

def simulation_1():
    L = 3  # number of interarrival times
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = G.rvs(L)
    print(a)

simulation_1()
```

s.1.12. Add this code to the other code and run it.

```
def simulation_2():
    N = 1  # number of customers
    L = 300

    G = uniform(loc=4, scale=2)
    a = G.rvs(L)

    plt.hist(a, bins=int(L / 20), label="a", density=True, cumulative=True)
    plt.title("N = {}, L = {}".format(N, L))

    x, y = cdf(a)
    plt.plot(x, y, label="d")
    plt.legend()
    plt.show()

simulation_2()
```

s.1.13. Add this to the other code and run it.

```
def KS(X, F):
    # Compute the Kolmogorov-Smirnov statistic where
    # X are the data points
    # F is the theoretical distribution
    support, y = cdf(X)
    y_theo = np.array([F.cdf(x) for x in support])
    return np.max(np.abs(y - y_theo))

def simulation_3():
    N = 1  # number of customers
    L = 300

    G = uniform(loc=4, scale=2)
    a = G.rvs(L)

    print(KS(a, G))

simulation_3()
```

s.1.14. Add this to the other code and run it.

```
from scipy.stats import expon
```

```
def plot_distributions(x, y, N, L, dist, dist_name):
    # plot the empirical cdf and the theoretical cdf in one figure
    plt.title("X ~ {} with N = {}, L = {}".format(dist_name, N, L))
    plt.plot(x, y, label="empirical")
    plt.plot(x, dist.cdf(x), label="exponential")
    plt.legend()
    plt.show()

def simulation_4():
    N = 1  # number of customers
    L = 300

    labda = 1.0 / 5  # lambda is a function in python. Hence we write labda
    E = expon(scale=1.0 / labda)
    print(E.mean()) # to check that we chose the right scale
    a = E.rvs(L)

    print(KS(a, E))
    x, y = cdf(a)
    dist_name = "U[4,6]"

    plot_distributions(x, y, N, L, E, dist_name)

simulation_4()
```

It is pretty obvious why these graphs must be different: we compare a uniform and an exponential distribution.

s.1.15. The following steps in code explain the logic.

```
def compute_arrivaltimes(a):
    A=[0]
    i = 1
    for x in a:
        A.append(A[i-1] + x)
        i += 1
    return A

def shop_1():
    a = [4, 3, 1.2, 5]
    b = [2, 0.5, 9]
    A = compute_arrivaltimes(a)
    B = compute_arrivaltimes(b)

    times = [0] + sorted(A[1:] + B[1:])
    print(times)

shop_1()
```

s.1.17. Add this to the other code and run it.

```
def shop_3():
    N, L = 3, 100
    G = uniform(loc=4, scale=2)
    a = superposition(G.rvs((N, L)))
```

```
    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))
    print(E.mean())

    x, y = cdf(a)
    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))  # Compute KS statistic using the function defined earlier

shop_3()
```

s.1.18. Add this to the other code and run it.

```
def shop_4():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))
    print(E.mean())

    x, y = cdf(a)
    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

shop_4()
```

This is great. Already for $N = 10$ patients we see that the exponential distribution is a real good fit for the inter-arrival times as observed by the doctor.

s.1.19. Add this to the other code and run it.

```
from scipy.stats import norm

def shop_5():
    N, L = 10, 100
    G = norm(loc=5, scale=1)
    a = superposition(G.rvs((N, L)))

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    x, y = cdf(a)
    dist_name = "N(5,1)"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

shop_5()
```

Clearly, whether the distribution of inter-arrival times of an individual patient are uniform, or normal, doesn't really matter. In both cases the exponential distribution is a good model for what the doctor sees. We did not analyze what happens if we would merge patients with normal distribution and uniform distribution, but we suspect that in all these cases the merged process converges to a set of i.i.d. exponentially distributed random variables. [12]

s.1.20. Since $\sigma = \mu = 5$, about 15% of the 'inter-arrival' times should be negative. This is clearly impossible.

```
# keep the numbering
```

s.1.21. Here is the answer.

```
def memoryless_1():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 0.5 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_1()
```

s.1.22. Here is the answer.

```
def memoryless_2():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 2 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_2()
```

s.1.23. Here is the answer.

[12] For the mathematically inclinded, can we prove such a property.

```
def memoryless_3():
    N, L = 10, 100_000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 2 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_3()
```

s.1.24. Suppose we would have set $s = 6$. Since the inter-arrival times of an individual patient are $\sim U[4,6]$, there cannot be any inter-arrival that is larger than 6. Thus, we cannot expect that the tail of the expontial distribution (which has an infinite support) will be a good model for the tail of our emperical distribution. In general, (as far as I know) using data to estimate tail probabilities is pretty sensitive, and one should be very careful to use data for this.

Also, it is important to know how data is obtained (and stored) before one starts with data analysis.

```
# keep the numbering
```

s.1.25. In step 1 we compute the arrival times of individual patients. In step 2 we merge these arrival times into a sorted set of arrival times as seen by the doctor. For step 3 I specify the mean first. Then, I want that each bin contains about this mean number of arrivals. Since there are NL arrivals in total, I want the number of bins to be equal to `NL/mean`.

The documentation of `np.bincount` and `np.histogram` explain how to use these functions.

s.1.26. For `mean=5` we do get acceptable results.

```
def poisson_2():
    N, L = 10, 1000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    # G = norm(loc=5, scale=1)
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2

    mean = 5  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
```

```
    # plt.pause(10)
    plt.show()

poisson_2()
```

s.1.27. Run this code to see the resemblance we have been looking for. It seems that our code satisfies this important consistency check.

```
def poisson_3():
    N, L = 10, 1000
    labda = 5
    G = expon(scale=1.0 / (N * labda))
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2

    mean = 20  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
    # plt.pause(10)
    plt.show()

poisson_3()
```

s.1.28. Here are some ideas you should have learned.

1. Algorithmic thinking, i.e., how to chop up a computational challenge into small steps.
2. An efficient method to compute the empirical distribution function
3. The Kolmogorov-Smirnov statistic
4. The empirical distribution of a merged inter-arrival arrival process converges, typically, super fast to an exponential distribution. Thus, inter-arrival times at doctors, hospitals and so on, are often very well described by an exponential distribution with suitable mean.
5. The convergence is not so sensitive to the distribution of the inter-arrival times of a single patients. For the doctor, only the population matters.
6. Using functions (e.g., `def compute(a)`) to document code (by the function name), hide complexity, and enables to reuse code so that it can be applied multiple times. Moreover, defining functions is in line with the extremely important Don't-Repeat-Yourself (DRY) principle.
7. Coding skills: python, `numpy` and `scipy`.
8. Data analysis is tricky, and you need different approaches to obtain information about the data.

s.2.1. We need a few imports.

```
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

plt.ion() # to skip making the graphs, I only use it for testing purposes.

scipy.random.seed(3)

def compute_Q_d(a, s, q0=0):
    # a: no of arrivals in a period
    # s: no of services at start of a period
    d = np.zeros_like(a) # departures
    Q = np.zeros_like(a) # queue lengths
    Q[0] = q0 # starting level of the queue
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = Q[i-1] + a[i] - d[i]

    return Q, d
```

Hopefully you notice how close the python code resembles the maths.

s.2.3. You should see that the queue starts at 100 and drains roughly at rate $\lambda - \mu$. If you make Q_0 very large (10000 or so), you should see that the queue length process behaves nearly like a line until it hits 0.

s.2.4. This is the code.

```
def cdf(a):
    y = range(1, len(a) + 1)
    y = [yy / len(a) for yy in y]  # normalize
    x = sorted(a)
    return x, y

def experiment_3():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)

    print(d.mean())

    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()

experiment_3()
```

s.2.5. The mean number of departures must be (about) equal to the mean number arrivals per period. Jobs cannot enter from 'nowhere'.

```
# A void cell for the numbering
```

s.2.6. The queue length drains at rate $\mu - \lambda$ when q_0 is really large. Thus, for such settings, you might just as well approximate the queue length behavior as $q(t) = q_0 - (\mu - \lambda)t$, i.e., as a deterministic system.

s.2.7. Copy this code and run it. I don't use the values for N and $Q0$ as specified in the exercise; running the code becomes somewhat slow, and this gets on my nerves while testing everything. Hence, you should replace the numbers by the ones mentioned in the exercise.

```
def experiment_5():
    N = 10  # set this to the correct value.
    labda = 6
    mu = 5
    q0 = 30

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)

    plt.plot(Q)
    plt.show()


experiment_5()
```

You should see that the queue drains with relatively less variation. The 'line' becomes 'straighter'.

s.2.8. Copy this code and run it.

```
def experiment_6():
    N = 100 #  Again, replace the numbers
    labda = 5
    mu = 6
    q0 = 10

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6()
```

The queue should increase with rate $\lambda - \mu$. Of course the queue length process will fluctuate a bit around the line with slope $\lambda - \mu$, but the line shows the trend.

s.2.9. Here is the code.

```
def experiment_6a():
    N = 10 # Use a larger value here
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = np.ones_like(a) * mu

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6a()
```

You should observe that the mean and the variability of the queue length process decrease.

s.2.10. Copy this code and run it.

```
def experiment_6b():
    N = 10 # Change to a larger value.
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(1.1 * mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6b()
```

For our experiments it is about 20% extra.

s.2.11. Here is one implementation.

```
def compute_Q_d_with_extra_servers(a, q0=0, mu=6, threshold=np.inf, extra=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False  # extra employees are not in
    for i in range(1, len(a)):
        rate = mu + extra if present else mu  # service rate
        s = poisson(rate).rvs()
        d[i] = min(Q[i - 1], s)
        Q[i] = Q[i - 1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= threshold:
            present = True  # hire employee for next period

    return Q, d


def experiment_7():
    N = 100 # take a big number here.
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)

    Q, d = compute_Q_d_with_extra_servers(a, q0, mu=6, threshold=20, extra=2)
    print(Q.mean(), Q.std())

    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()

experiment_7()
```

Note that this code runs significantly slower than the other code. This is because we now have to call `poisson(mu).rvs()` in every step of the for loop. We could make the program run faster by using the `scipy` library to generate N random numbers in one step outside of the for loop and then extract numbers from there when needed. However, for the sake of clarity we chose not to do so.

s.2.12. Here is an example. What is the meaning of `np.inf`?

```
def compute_Q_d_blocking(a, s, q0=0, b=np.inf):
    # b is the blocking level.
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    for i in range(1, len(a)):
        d[i] = min(Q[i - 1], s[i])
        Q[i] = min(b, Q[i - 1] + a[i] - d[i])

    return Q, d



def experiment_7a():
    N = 100 # take a larger value
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d_blocking(a, s, q0, b=15)
    print(Q.mean(), Q.std())

    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()


experiment_7a()
```

s.2.13. Here is all the code.

```
def compute_cost(a, mu, q0=0, threshold=np.inf, h=0, p=0, S=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False  # extra employee is not in.
    queueing_cost = 0
    server_cost = 0
    setup_cost = 0
    for i in range(1, len(a)):
        if present:
            server_cost += p
            c = poisson(mu).rvs()
        else:
            c = 0  # server not present, hence no service
```

```
        d[i] = min(Q[i - 1], c)
        Q[i] = Q[i - 1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= threshold:
            present = True  # switch on server
            setup_cost += S
        queueing_cost += h * Q[i]

    print(queueing_cost, setup_cost, server_cost)

    total_cost = queueing_cost + server_cost + setup_cost
    num_periods = len(a) - 1
    average_cost = total_cost / num_periods
    return average_cost


def experiment_8():
    N = 100
    labda = 0.3
    mu = 1
    q0 = 0
    theta = 100  # threshold

    h = 1
    p = 5
    S = 500

    a = poisson(labda).rvs(N)
    av = compute_cost(a, mu, q0, theta, h, p, S)
    print(av)

experiment_8()
```

After a bit of experimentation, we see that $\theta = 15$ is quite a bit better than $\theta = 100$.

s.2.14.

```
def experiment_9():
    N = 10 # Change to a larger value.
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)
    loss = (Q > 20)
    print(loss.sum())
    total_demand = a.sum()
    print(100*(Q > 0).sum()/total_demand)


experiment_9()
```

I multiply with 100 to get a percentage.

s.2.15. Some important points are the following.

1. Making a simulation requires some ingenuity, but is need not be difficult.
2. It is often much simpler to analyze difficult queueing situations with simulation than with mathematics. In fact, most queueing systems cannot be analyzed with maths.
3. We studied the behavior of queues under certain control policies, typically policies that change the service rate as a function of the queue length. Such policies are used in practice, such as in supermarkets, hospitals, the customs services at airports, and so on. Hence, you now have some tools to design and analyze such systems.

An interesting extension is to incorporate time-varying demand. In many systems, such as supermarkets, the demand is not constant over the day. In such cases the planning of servers should take this into account.

s.3.1. If you have read (and thought about) the documentation on the web, you must have ended up with this code.

```
from heapq import heappop, heappush
import scipy
from scipy.stats import uniform, expon

scipy.random.seed(3)

def sort_ages():
    stack = []

    heappush(stack, (21, "Jan"))
    heappush(stack, (20, "Piet"))
    heappush(stack, (18, "Klara"))
    heappush(stack, (25, "Cynthia"))

    while stack:
        age, name = heappop(stack)
        print(name, age)

sort_ages()
```

heappush pushes things on the stack in a sorted fashion, heappop takes things from the stack. First we put the students on the stack. To print the student names in sorted order of agee, we remove items from the stack until it is empty.

s.3.2. We can make the tuples, i.e., the data between the brackets, just longer.

```
def sort_ages_with_more_info():
    stack = []

    heappush(stack, (21, "Jan", "Huawei"))
    heappush(stack, (20, "Piet", "Apple"))
    heappush(stack, (18, "Klara", "Motorola"))
    heappush(stack, (25, "Cynthia", "Nexus"))

    while stack:
        age, name, phone = heappop(stack)
        print(age, name, phone)

sort_ages_with_more_info()
```

Observe that we label the students by their age, and organize the data in the form of tuples.

s.3.3. For a queueing process we can start with putting a number of job arrivals on the stack and label these events as 'arrivals'. Whenever a service starts, compute the departure time of a job, and put this departure moment on the stack. Label this event as a 'departure'. Then move to the next event. Since the event stack is sorted in time, the event at the head of the stack is the first moment in time something useful happens.

More generally, a discrete-time stochastic process moves from event to event. At an event certain actions have to be taken, and these actions may involve the generation of new events or the removal of events that are in the stack. The new events are put on the stack, the obsolete ones removed from the stack, and then the simulator moves to the first event on the stack, actions are taken, move to the next event, and so on.

```
# void  code for numbering
```

s.3.4. We use variables for clarity and ease of reading the code. We will simply make less mistakes like this.

s.3.5. The wording is important here. Objects are instances of classes. As an example: Albert Einstein was a human being. Here, 'Einstein' is the object, and 'human being' is a class.

s.3.6. One way is like this.

```
def experiment_1():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon

    num_jobs = 10

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        print(job)



experiment_1()
```

s.3.7. We need a server object to keep track of the state of the server, and we also need a list to queue the jobs.

```
class Server:
    def __init__(self):
        self.busy = False

server = Server()
```

```
queue = []
served_jobs = [] # used for statistics

def start_service(time, job):
    server.busy = True
    job.departure_time = time + job.service_time
    heappush(stack, (job.departure_time, job, DEPARTURE))

def handle_arrival(time, job):
    job.queue_length_at_arrival = len(queue)
    if server.busy:
        queue.append(job)
    else:
        start_service(time, job)

def handle_departure(time, job):
    server.busy = False
    if queue: # queue is not empty
        next_job = queue.pop(0) # pop oldest job in queue
        start_service(time, next_job)
```

s.3.8. Here is an experiment.

```
def experiment_2():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon
    num_jobs = 10  # too small, change it to a larger number, and rerun the experiment

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", rho * rho / (1 - rho))
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn/len(served_jobs)
    print("Theoretical avg. sojourn time:", (1./labda)*rho/(1-rho))
```

```
    print("Simulated avg. sojourn time:", av_sojourn_time)

experiment_2()
```

s.3.9. Now we see that we needed to store the jobs in `served_jobs`. Set `num_jobs=100` in the code of Exercise 3.6. Put the following code at the end of the simulator to compute the statistics.

```
def experiment_2a():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon
    num_jobs = 1000

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", rho * rho / (1 - rho))
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn/len(served_jobs)
    print("Theoretical avg. sojourn time:", (1./labda)*rho/(1-rho))
    print("Simulated avg. sojourn time:", av_sojourn_time)

experiment_2a()
```

Now run the simulator.

You should see that you need a real large amount of jobs to obtain a good estimate for the (theoretical) expected queue length[13].

s.3.10. With `F=uniform(3, 0.00001)` the job inter-arrival times are nearly 3. Likewise, the service times are nearly 2. Hence, arriving customers will always see an empty queue, implying that the average queue length upon arrival equals zero. When you reverse the 2 and 3, the queue should build up.

s.3.11. Here is simple implementation.

[13] I have to admit that I don't understand why (at least in my simulation) I need about 10^5 jobs to get a good estimate.

```
def pollaczek_khinchine(labda, G):
    ES = G.mean()
    rho = labda*ES
    ce2 = G.var()/ES/ES
    EW = (1.+ce2)/2 * rho/(1-rho)*ES
    return EW


labda = 1./3
F = expon(scale=1./labda)  # interarrival time distributon
G = uniform(1, 2)

print("PK: ", labda*pollaczek_khinchine(labda,G))
```

s.3.12. To ensure that we do not keep old data, we have to reset all lists.

```
stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def test_mg1():
    job = Job()
    labda = 1.0 / 3
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = uniform(1, 3)
    print("ES: ", G.mean(), "rho: ", labda * G.mean())

    num_jobs = 1000

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", labda * pollaczek_khinchine(labda, G))
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Theoretical avg. sojourn time: ", pollaczek_khinchine(labda, G) + G.mean())
    print("Avg. sojourn time:", av_sojourn_time)
```

```
test_mg1()
```

We used Little's law to relate the average queue length and the average waiting time.

s.3.13. We simulate 60 uniformly distributed arrival times $\sim U[0,120]$. Then we sort them so that the customers arrive in order and store them in an array.

```
import numpy as np

F = np.sort(uniform(0, 120).rvs(60))
```

s.3.14. We need to start with importing the Counter class.

```
from collections import Counter

stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def check_in():
    num_jobs = 60
    F = np.sort(uniform(0, 120).rvs(num_jobs))
    G = uniform(1, 3)

    for i in range(num_jobs):
        job = Job()
        job.arrival_time = F[i]
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Avg. sojourn time:", av_sojourn_time)

    c = Counter([j.queue_length_at_arrival for j in served_jobs])
    print("Queue length distributon, sloppy output")
    print(sorted(c.items()))


check_in()
```

s.3.15. In this specific sample path, most jobs appear to arrive at the end of the 2 hours. And indeed, when all people arrive late, we'll see a big queue.

s.3.16. Here is the code.

```
stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def check_in_2():
    num_jobs = 60
    F = np.sort(uniform(0, 150).rvs(num_jobs))
    G = uniform(1, 2)

    for i in range(num_jobs):
        job = Job()
        job.arrival_time = F[i]
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Avg. sojourn time:", av_sojourn_time)

    x = [job.arrival_time for job in served_jobs]
    y = [job.queue_length_at_arrival for job in served_jobs]

    plt.plot(x, y, "o")
    plt.show()
check_in_2()
```

This is much better.

s.3.17. This is really easy: change the line `queue.pop(0)` by `queue.pop()`.

s.3.18. As in the LIFO example, we only have to change the data structure. First, we give each job an extra attribute corresponding to its priority. Then we use a heap to store the jobs in queue. Specifically, change the line with `queue.append(job)` by `heappush(queue, (job.priority, job))` and `queue.pop(0)` by `heappop(queue)`.

s.3.19. In a python list the `pop(0)` function is an $O(n)$ operation, where n is the number of elements in the list. In a deque appending and removing items to either end of the deque is an $O(1)$ operation.

When you do large scale simulations, involving many hours of simulation time, all inefficiencies build up and can make the running time orders of magnitude longer. A famous example is the sorting of numbers. A simple, but stupid, sorting algorithm is $O(n^2)$ while a good algorithm is $O(n \log n)$. The sorting time of 10^6 numbers is dramatically different.

For our code, add the line

```
from collections import deque
```

and replace `queue = []` by

```
queue = deque()
```

and `queue.pop(0)` by

```
queue.popleft()
```

s.3.20. Typically, generating all jobs at the start is not a real good idea. When running large simulations the amount of computer memory required to store all this data grows out of hand. Also our `served_jobs` list is not memory efficient.

s.3.21. Topics learned.

1. Efficient data structures such as heap queues. In general, it is important to have some basic knowledge of algorithmic complexity.
2. Event stacks to organize the tracking of events in time. With the concept of event stack, you now know how discrete event simulators work.
3. The simulation of the $G/G/1$ queue. We can use simulation to analyze a practical case and come up with concrete recommendations on how to improve the system.

s.3.22. It would be interesting to change the demand process such that the arrival rate becomes time-dependent. That would make the case more realistic, and would make the simulator more useful. In fact, the theoretical models nearly always assume that the arrival and service time distribution are constant. When these become dynamic, simulation is the way to go.

s.4.1. If the number of busy servers is less than c, a service can start. Otherwise a job has to queue. If a service starts, increase `num_busy` by one.

s.4.2. At a departure, a server becomes free, hence decrease `num_busy` by one. If there are jobs in queue, start a new service.

s.4.3. Of course, the number of busy servers cannot be negative or larger than c. The queue length cannot be negative. Finally, it cannot be that the queue length is positive and the number of busy servers is less than c.

s.4.4. See `tutorial_4.py`. Study it in detail so that you really understand how it works.

s.4.5. First we instantiate, then we run and print.

```
labda = 2
mu = 1
ggc = GGc(expon(scale=1./labda), expon(scale=1./mu), 3, 10)
ggc.run()

print(ggc.served_jobs)
```

s.4.7. Recall that Sakasegawa's formula is exact for the $M/G/1$ queue, hence also for the $M/M/1$ queue.

```
def sakasegawa(F, G, c):
    labda = 1./F.mean()
    ES = G.mean()
    rho = labda*ES/c
    EWQ_1 = rho**(np.sqrt(2*(c+1)) - 1)/(c*(1-rho))*ES
    ca2 = F.var()*labda*labda
    ce2 = G.var()/ES/ES
    return (ca2+ce2)/2 * EWQ_1
```

s.4.8. In the next function we perform the test. We give this function standard arguments so that we can run the function as a standalone test. As such we want to to contain all parameter values.

```
def mm1_test(labda=0.8, mu=1, num_jobs=100):
    c = 1
    F = expon(scale=1./labda)
    G = expon(scale=1./mu)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/M/1 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G, c))
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

mm1_test(num_jobs=100)
mm1_test(num_jobs=10000)
```

s.4.9. Here is a function to keep things organized.

```
def md1_test(labda=0.9, mu=1, num_jobs=100):
    c = 1
    F = expon(scale=1./labda)
    G = uniform(mu, 0.0001)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/1 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G, c))
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

md1_test(num_jobs=100)
md1_test(num_jobs=10000)
```

s.4.10. The code.

```
def md2_test(labda=1.8, mu=1, num_jobs=100):
    c = 2
    F = expon(scale=1./labda)
    G = uniform(mu, 0.0001)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/2 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G, c))
    print("Simu avg. waiting time in queue:", avg_wait_in_q)
```

```
md2_test(num_jobs=100)
md2_test(num_jobs=10000)
```

s.4.11. Since most people arrive independent of each other, it is reasonable that the arrival process in this period of duration T is Poisson with rate $\lambda = N/T$.

s.4.12. It is reasonable *within the context* that the mean waiting time is less important than the largest waiting time and the fraction of people that wait longer than 10 minutes, say. Note here, *context is crucial* when making models.

s.4.13. There are, by assumption, only arrivals between 9 and 10, hence the arrival process is not stationary, hence the queueing process is not stationary. It is also not a problem if the system is temporarily in overload. In fact, as long as c is such that the performance measures (maximum delay and fraction of people waiting longer than some threshold) are within reasonable bounds, the system works as it should.

s.4.14. We study the dynamics of a queueing process in which the demand grows and than peters out. Such queueing situations are, typically, mathematically intractable. In fact, we finally have come to a point in which we really need simulation.

s.4.15. The first function computes the KPIs, the second runs multiple scenarios.

```python
def intake_process(F, G, c, num):

    ggc = GGc(F, G, c, num)
    ggc.run()

    max_waiting_time = max(j.waiting_time() for j in ggc.served_jobs)
    longer_ten = sum((j.waiting_time()>=10)for j in ggc.served_jobs)
    return max_waiting_time, longer_ten


def intake_test_1():
    labda = 300/60
    F = expon(scale=1.0 / labda)
    G = uniform(1, 2)
    num = 300

    print("Num servers, max waiting time, num longer than 10")
    for c in range(3, 10):
        max_waiting_time, longer_ten = intake_process(F, G, c, num)
        print(c, max_waiting_time, longer_ten)


intake_test_1()
```

s.4.16. I get these numbers

```
Num servers, max waiting time, num longer than 10
3 138.1697966325396 280
4 82.7792212258584 266
5 54.741743193174834 227
6 39.35432320034931 220
7 22.868514027211248 167
8 15.77953068102207 102
9 12.118348481100451 50
```

Even for 9 servers, 50 people wait longer than 10 minutes. Interestingly, with 9 servers the maximum waiting time is just 12 minutes, so we can at least keep this within bounds.

We could also make a plot of the job waiting time as a function of time, but we skip it here.

Perhaps we should open the check-in desks for a longer time and advise people to come earlier. Let's try 90 minutes instead.

```
def intake_test_2():
    labda = 300/90
    F = expon(scale=1.0 / labda)
    G = uniform(1, 2)
    num = 300

    print("Num servers, max waiting time, num longer than 10")
    for c in range(3, 10):
        max_waiting_time, longer_ten = intake_process(F, G, c, num)
        print(c, max_waiting_time, longer_ten)
```

If you do the experiment, you'll see that this works much better. It is now also very simple to set up a third experiment in which the desks are open during two hours. Then you'll see that 4 servers suffice.

s.4.18. Topics learned.

1. The simulation of the $G/G/c$ queue
2. Python classes and, a bit more generally, object oriented programming.
3. Organizing cases and experiments by means of functions. With these functions we can keep a log of what we precisely tested, what parameter values we used and which code we ran. These tests are also useful to show how to actually use/run the code.

Some interesting extensions.

1. Scale up to networks of $G/G/c$ queues.
2. In the $G/G/c$ queue the assumption is that all servers have the same service rate. In production settings this is typically not the case. There is a queue of jobs, and these jobs are served by machines with different speeds. For instance, old machines may work at a slower rate than new machines.

s.5.1. Average waiting time of each job class, and maximal waiting time of each class. We might also be interested in the fraction of jobs spending less than 10 minutes in the system.

s.5.2. Here are two tests. The tests will not yet work since we have not yet made the class to simulate the queueing system. In other words, we know that our first test should fail initially.

```
def DD1_test_1():
    # test with only business customers
    c = 0
    F = uniform(1, 0.0001)
    G = expon(0.5, 0.0001)
    p_business = 1
    num_jobs = 5
    jobs = generate_jobs(F, G, p_business, num_jobs)
    ggc = GGc_with_business(c, jobs)
    ggc.run()
    ggc.print_served_job()
```

```
def DD1_test_2():
    # test with only economy customers
    c = 1
    F = uniform(1, 0.0001)
    G = expon(0.5, 0.0001)
    p_business = 0
    num_jobs = 5
    jobs = generate_jobs(F, G, p_business, num_jobs)
    ggc = GGc_with_business(c, jobs)
    ggc.run()
    ggc.print_served_job()

def do_tests():
    DD1_test_1()
    #DD1_test_2()

do_tests()
```

s.5.3. The code explains all. Note, we will not use it in our simulation.

```
def generate_jobs_bad_implementation(F, Ge, Gb, p_business, num_jobs):
    # the difference in performance is tremendous
    for n in range(num_jobs):
        job = Job()
        job.arrival_time = time + F.rvs()
        if uniform(0,1).rvs() < p_business:
            job.customer_type = BUSINESS
            job.service_time = Gb.rvs()
        else:
            job.customer_type = ECONOMY
            job.service_time = Ge.rvs()

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

s.5.9.

1. First, check whether the our simulator is indeed correct. Is there a specific queue for business class customers? If so, do the business server(s), when idle, indeed take economy class customers in service?

2. You'll need real data to see whether our simple arrival process and service distributions are realistic. I suppose most of this is already known. For instance, the number of customers on the plain is known, as is the number of business customers. Service times can obtained from the desks, either by measuring, or by using the times the boarding passes are printed.

3. Once you have good data, vary one parameter at a time, and report the results *graphically*. People typically want to see trends; graphs are indispensable for this.

Finally, you often need to compare a host of different policies:

- Perhaps it is better to have dynamic service capacity, 3 during the first hour, 6 in the second.
- Perhaps the service distributions of the business customers can be made a bit shorter, or more regular. What would be impact of this?

- Since there is small number of business customers, why not open the business check-in desk later than the other desks? Or, share the desks during the first opening hour of the desks, and 'unshare' during the second?